Computer Security (COM-301)

Monday live exercises

Execution Attacks and Security testing

Carmela Troncoso
SPRING Lab
carmela.troncoso@epfl.ch

DEP protection

The startup `NewHaven` hires you to help them develop secure software.

The `NewHaven` developers tell you that in their server, they sometimes need to execute code that is provided as a parameter in a function. But they tell you that this is safe because the server implements Data Execution Prevention. Is this correct? (Justify your answer).

It is not correct: If DEP is implemented and active; anything that is written in the stack (*such as a parameter*) cannot be executed.

DEP protection

The startup `NewHaven` hires you to help them develop secure software.

The `NewHaven` developers tell you that in their server, they sometimes need to execute code that is provided as a parameter in a function. But they tell you that this is safe because the server implements Data Execution Prevention. Is this correct? (Justify your answer).

Answer

It is not correct: If DEP is implemented and active; anything that is written in the stack (*such as a parameter*) cannot be executed.

It is not correct: If DEP is implemented and active; anything that is written in the stack (*such as a parameter*) cannot be executed.

The (true or not) power of the canary

(a)Stack canaries, which are a defense against buffer overflow attacks, require operating system support

(a)Stack canaries will prevent any attack that overflows local variables from executing injected code

(a)Stack canaries protect against all printf format string vulnerabilities

(a) False. A compiler can add the canary generation and checking without needing any extra functionality from the operating system. Here more info on how this works: https://ctf-wiki.github.io/ctf-wiki/pwn/linux/mitigation/canary/#canary-implementation-principle

(a) False. The canary does not avoid *any* attack. If the attack enables the canary to be overwriten or skipped, then the canary will not help.

(a) False. Only if the use of the format string overwrites the canary. Forms of format string vulnerabilities read data from the stack but do not alter stack contents; or write onto the stack in pinpoint locations (for example, to alter the value of a variable) cannot be prevented by a canary.

The (true or not) power of the canary

(a) Stack canaries, which are a defense against buffer overflow attacks, require operating system support
(b) Stack canaries will prevent any attack that overflows local variables from executing injected code
(c) Stack canaries protect against all printf format string vulnerabilities

Answer
(a)False. A compiler can add the canary generation and checking without needing any extra functionality from the operating system. Here more info on how this works: https://ctf-wiki.github.io/ctf-wiki/pwn/linux/mitigation/canary/#canary-implementation-principle
(b)False. The canary does not avoid *any* attack. If the attack enables the canary to be overwriten or skipped, then the canary will not help.
(c)False. Only if the use of the format string overwrites the canary. Forms of format string vulnerabilities read data from the stack but do not alter stack contents; or write onto the stack in pinpoint locations (for example, to alter the value of a variable) cannot be prevented by a canary.

(a) False. A compiler can add the canary generation and checking without needing any extra functionality from the operating system. Here more info on how this works: https://ctf-wiki.github.io/ctf-wiki/pwn/linux/mitigation/canary/#canary-implementation-principle

(a) False. The canary does not avoid *any* attack. If the attack enables the canary to be overwriten or skipped, then the canary will not help.

(a) False. Only if the use of the format string overwrites the canary. Forms of format string vulnerabilities read data from the stack but do not alter stack contents; or write onto the stack in pinpoint locations (for example, to alter the value of a variable) cannot be prevented by a canary.

Bypassing the password

Consider the following C function for getting a password from the standard input and checking it (strcmp compares two strings, and returns 0 if strings are equal, or a number that is not equal to zero if they are not):

```
getPassword() {
    int isCorrect = 0;
    char password[12];

    printf("Enter your password > ");
    gets(password);

    if(strcmp(password, "C0m301-Adm1N") == 0) isCorrect = 1;

    return isCorrect;
}
```

Consider the following fuzzing strategy: a fuzzer generates and feeds strings composed of multiple '\0' characters (null string terminators), of length up to 15.
Is this strategy able to uncover a vulnerability? If yes, identify which vulnerability and explain why the fuzzer finds it. If not, propose an alternative approach to test (may or may not be fuzzing) that uncovers a vulnerability. Explain why your approach would find that vulnerability.

This strategy will not be able to uncover a vulnerability.

The vulnerability is: The length of the input taken by the gets function is not verified so password could take more than the allocated stack memory.

The fuzzer will not uncover the vulnerability
This will at most overwrite parts of the isCorrect variable and not the return address. But the overwrite will be with 0.
The comparison strcmp will not succeed, so isCorrect will be zero (as in not knowing the password)

A fix would be to change the fuzzing to 1s, which would result on isCorrect being True even not knowing the password.

Bypassing the password

Consider the following C function for getting a password from the standard input and checking it (strcmp compares two strings, and returns 0 if strings are equal, or

```
getPassword() {
    int isCorrect = 0;
    char password[12];

    printf("Enter your password > ");
    gets(password);

    if(strcmp(password, "C0m301-Adm1N") == 0)

    return isCorrect;
}
```

Consider the following fuzzing strategy: a fuzzer generates
(null string terminators), of length up to 15.
Is this strategy able to uncover a vulnerability? If yes, iden
it. If not, propose an alternative approach to test (may or
Explain why your approach would find that vulnerability.

Answer
This strategy will not be able to uncover a vulnerability.

The vulnerability is: The length of the input taken by the gets function is not verified so password could take more than the allocated stack memory.

The fuzzer **will not uncover the vulnerability**
This will at most overwrite parts of the isCorrect variable and not the return address. But the overwrite will be with 0.
The comparison strcmp will not succeed, so isCorrect will be zero (as in not knowing the password)

A fix would be to change the fuzzing to 1s, which would result on isCorrect being True even not knowing the password.

Bonus for everyone

Gru has written the code in the next slide to manage the bonuses of minions that participate in evil missions. This function receives an identifier for a minion and then prompts the minion to provide the name of the mission they participated in. Gru asks for your help debugging the function.

Identify two lines that contain unsafe code that may lead to a memory safety error. For those two lines i) explain the vulnerability, and ii) explain whether a Minion can exploit this vulnerability to increase their bonus even when their mission did not succeed.

Assume that Minions are good teammates and will never steal a bonus from another Minion by providing others' successful mission names

Assume that local variables are pushed to the pile as in the order they are created

Bonus for everyone

Identify two lines that contain unsafe code that may lead to a memory safety error. For those two lines i) explain the vulnerability, and ii) explain whether a Minion can exploit this vulnerability to increase their bonus even when their mission did not succeed.

```
int bonus[100] =  { 0 };                    /* array of 100 integers initialized to 0 */

char successDB(char* mission) {

    /* function that returns 0 for failed missions */
    /* and 1 for successful missions */
    /* if the mission does not exist, it crashes */

}

1: int AddMission(int minionID) {
2:      char success;                       /* one byte: 1 if mission succeeds, 0 otherwise */
3:      char mission[30];                   /* mission name */
4:
5:      gets(mission);                      /* reads mission name from keyboard */
6:      success = successDB(mission);       /* checks in the DB the success of the mission */
7:
8:      if (success == 1) { bonus[minionID] += 1 }; /* if mission succeeded, increase bonus */
9:
10:     return 0;
11: }
```

Line 5: Gets(mission) can overwrite success [no check in mission]

Success either is overwritten again in line 6; or the function successDB crashes.

Cannot get bonus, unless knows a number of a mission that has a bonus (which they won't do, recall they are nice minions)

Line 8: Lack of check on minion ID being <100

Would give an error, but not increase the bonus.

The only way of increasing bonus is, because there is no authentication (could be considered an error) they can give another minion ID and increase that minion's bonus. As this was not forbidden in the question we gave it as an ok answer.

Bonus for everyone

Identify two lines that contain unsafe code that may lead to a memory safety error. For those two lines i) explain the vulnerability, and ii) explain whether a Minion can exploit this vulnerability to increase their bonus even when their mission did not succeed.

```
int bonus[100] =  { 0 };                    /* array of 100 integ

char successDB(char* mission) {

    /* function that returns 0 for failed missions */
    /* and 1 for successful missions */
    /* if the mission does not exist, it crashes */

}

1: int AddMission(int minionID) {
2:      char success;                       /* one byte: 1 if miss
3:      char mission[30];                   /* mission name */
4:
5:      gets(mission);                      /* reads mission name
6:      success = successDB(mission);       /* checks in the DB th
7:
8:      if (success == 1) { bonus[minionID] += 1 }; /* if mission
9:
10:     return 0;
11: }
```

Answer

Line 5: Gets(mission) can overwrite success [no check in mission]

Success either is overwritten again in line 6; or the function successDB crashes.

Cannot get bonus, unless knows a number of a mission that has a bonus (which they won't do, recall they are nice minions)

Line 8: Lack of check on minion ID being <100

Would give an error, but not increase the bonus.

The only way of increasing bonus is, because there is no authentication (could be considered an error) they can give another minion ID and increase that minion's bonus. As this was not forbidden in the question we gave it as an ok answer.

Line 5: Gets(mission) can overwrite success [no check in mission]

Success either is overwritten again in line 6; or the function successDB crashes.

Cannot get bonus, unless knows a number of a mission that has a bonus (which they won't do, recall they are nice minions)

Line 8: Lack of check on minion ID being <100

Would give an error, but not increase the bonus.

The only way of increasing bonus is, because there is no authentication (could be considered an error) they can give another minion ID and increase that minion's bonus. As this was not forbidden in the question we gave it as an ok answer.

The more fuzz, the fuzzier

When you fuzz a program… True or false (as always, justify!)

[a] Using one fuzzer is enough

[b] You should use as many fuzzers as possible for as long as possible

[c] Running three complementary fuzzers for 1000 iterations each is enough

[d] It does not matter which fuzzer you choose. All types are good.

(a) False: there is no perfect fuzzer, so running one cannot be enough
(b) True: using different fuzzers that produce many different inputs for as long as possible ensures that one will get more coverage.
(c) This is only enough if those 1000 iterations of those fuzzers can achieve complete coverage (impossible for any commercial big program)
(d) Types matter: how inputs are chosen determines the code coverage that will be achieved.

The more fuzz, the fuzzier

When you fuzz a program… True or false (as always, justify!)

[a] Using one fuzzer is enough

[b] You should use as many fuzzers as possible for as long as possible

[c] Running three complementary fuzzers for 1000 iterations each is enough

[d] It does not matter which fuzzer you choose. All types are good.

Answer
(a)False: there is no perfect fuzzer, so running one cannot be enough
(b)True: using different fuzzers that produce many different inputs for as long as possible ensures that one will get more coverage.
(c)This is only enough if those 1000 iterations of those fuzzers can achieve complete coverage (impossible for any commercial big program)
(d)Types matter: how inputs are chosen determines the code coverage that will be achieved.

(a) False: there is no perfect fuzzer, so running one cannot be enough
(b) True: using different fuzzers that produce many different inputs for as long as possible ensures that one will get more coverage.
(c) This is only enough if those 1000 iterations of those fuzzers can achieve complete coverage (impossible for any commercial big program)
(d) Types matter: how inputs are chosen determines the code coverage that will be achieved.